Beep

The microservice way

Hugo Ponthieu

Table of Contents

Abstract

Beep is a chat application that allows users to communicate with each *other in real-time. The goal of this document is to provide an overview of the architecture of the application, including the services, protocols, and security mechanisms used to build the application.*

Currently the application is developed in a monolithic way. As the features grow, the application is harder to maintain and scale. In fact, all the elements of the application can only be scaled together. Futhermore, the application is not fault tolerant, if a service fails, the whole application will be down.

Those outage can be caused by high loads or network issues. It can also come from the development team that can introduce bugs in the application. The decoupling of responsibilities that offer microservices architecture will help to reduce the impact of a single component failure and let us to scale the application more easily.

1. Application Overview

1.1. Current features

Chatting via channels: Users can create channels and send messages to each other. They can also create private channels to communicate with their friends.

Servers: Server are the main entity of the application. They regroup users and channels. A user can be part of multiple servers. A server can have multiple channels. Inside, allowed users can communicate with each other or define roles. Web

1.2. End-user capabilities

To better design and implement the application, we will need to define the capabilities of the users. With the needs of client in mind, we will be able to define parts of the application that can be separated.

But also we will be able to define any point of dependency between the services. This will allow us to define potential interactions between the services.

We will try to define the capabilities of the user by making it impersonate different characters. There will be characters based on the authorization level of the user: - Guest user: A user that is not authenticated - User: A user that is authenticated

But also on more fine grained authorization level: - UserModerator: A user that can perform some actions on the users - ServerModerator: A user that can perform manage the server of the application

And of course there will be more fined grained authorization on resources that we will define along with the capabilities of the user.

1.3. Account

We will begin like if a user just discovers the application. He will be able to access the application as a guest user.

As a guest user I want to be able to sign up using:

- My email and password
- My google account
- My Polytech account

The users that are sign up will have the abilitie to authenticate and access the application only once their account is validated.

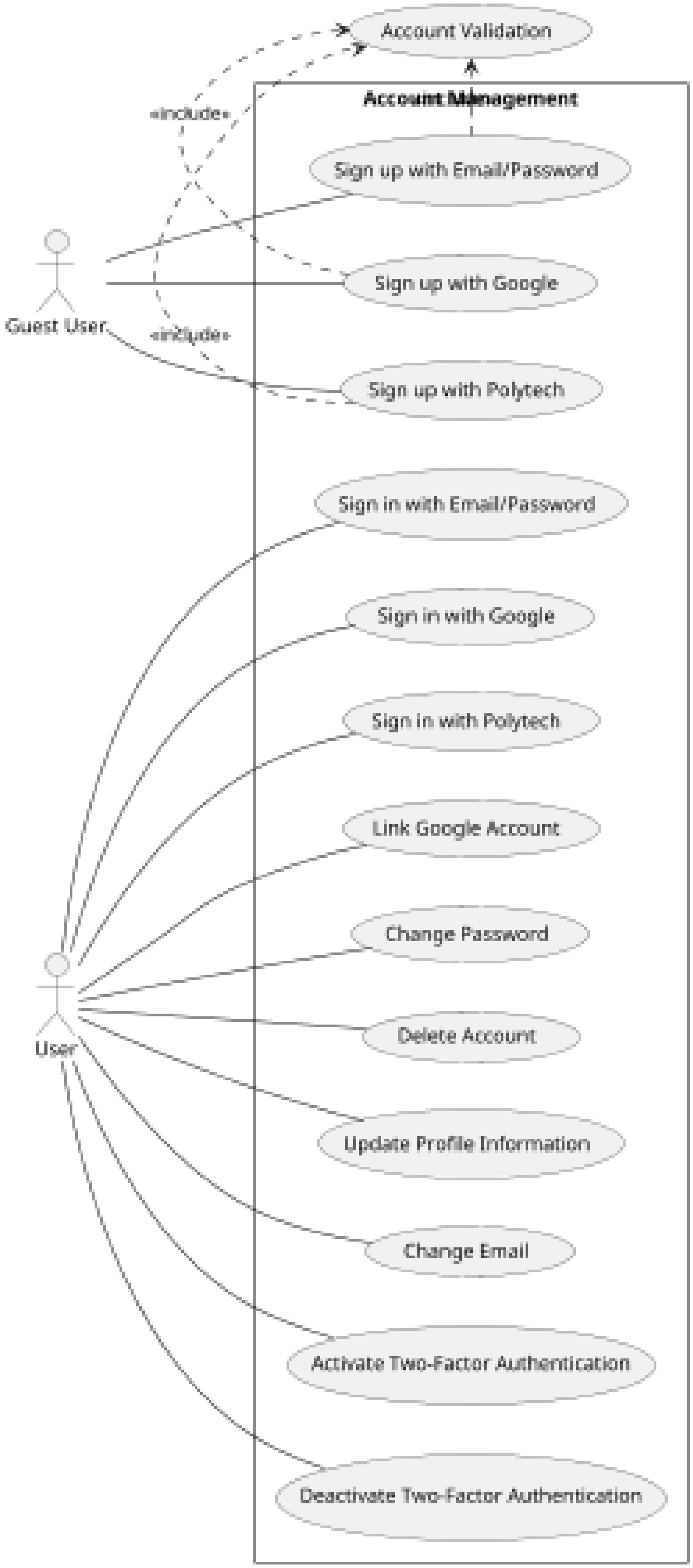
As a user:

- I want to be able to sign in using the same methods as the sign up in order to access the application.
- that is sign in with my email and password I want to be able to link my google account to my account so that I can later authenticate with it.
- I want to change my password in order to secure my account.
- I want to be able to delete my account in order to leave the application.

Once the user is authenticated he will be able to access the application. Therefore we will focus now more on action that he can perform on his profile information.

As a user:

- I want to be able to update my profile information such as my name, last name, nickname and profile picture in order to keep it up to date.
- I want to be able to change my email in order to transfer my account to another email.
- I want to be able to change my password in order to secure my account.
- I want to be able to activate Two-Factor Authentication in order to secure my account.
- I want to be able to deactivate Two-Factor Authentication in order to secure my account.
- I want to be able to delete my account in order to leave the application.


Account Validation
Account Management
«include»
«include»
Guest User
Sign up with Email/Password
Sign up with Google
Sign up with Polytech
Sign in with Email/Password
Sign in with Google
Sign in with Polytech
Link Google Account
Change Password
Delete Account
Update Profile Information
Change Email
Activate Two-Factor Authentication
Deactivate Two-Factor Authentication
User

1.3.1. Social

Once authenticated the user will be able to access some features without any further authorization. He will be able to access the friends system. We can first focus on the friend management for a given user.

As a user I want to:

- invite a user from their username to add a friend.
- list the invitations that I have sent or that other user sent me in order to manage them.
- accept a friend request in order to connect with the user.
- decline a friend request.
- cancel a friend request that I have sent in order to not have the user as friend.

Once I am friend with a user I want to be able to manage my frienship.

As a user I want to:

- list my friends in order to see who are my friends.
- remove a friend in order to not have him as friend anymore.

In order to regroup users, users be members of servers. There are 2 types of servers, public and private. The user can join a public server without any authorization. But he will need to be invited to join a private server. So as a user I want to:

- see all the public servers in order to join them.
- see all the servers that I am member of in order to manage them.
- leave server so that I am not related to it anymore.
- be able to answer to a server invitation so I can be a member of a server.
- browse the servers by their name and description so I can find the communities that I want to join.

NOTE: USE CASE DIAGRAM

1.3.2. Chatting

As user discover other users, he will want to interact with them. He will be able to do that through the chat system. It is composed of channel that contain messages. We will see in that part what are the abilities. As a user I want to:

- create a channel to be able to communicate with other users.
- delete a channel in order to not have it anymore.
- list the channels that I am part of in order to manage them.
- join a channel in order to communicate with the users.
- leave a channel in order to not be part of it anymore.

- add a user to a channel in order to let him communicate with the users.
- to search thrgouh the entire messages of a channel to find a message based on a keyword

With access to a channel the user will want to discuss with other users. As a user I want to:

- send a message in a channel in order to communicate with the users.
- send files in a message in order to share them with the users.
- delete a message so that I clean a channel.
- edit a message in order to correct it.
- list the messages of a channel in order to see the history of the channel.
- to pin messages in a channel to keep them visible for long time.

NOTE: USE CASE DIAGRAM

1.3.3. Severs

As cited before the user will be able to join servers. They regroup users and channels. A user that is authenticated and that as access to a particular server is called a member of the server.

By default a member will not perform any action on the server. He will need to be granted with a role to perform some actions. Role are defined at the server level and they will be aggregation of more fine-grained role.

The fine-grained roles will be:

- administrator
- server manager
- role manager
- channel manager
- channel viewer
- webhook manager
- nickname manager
- nickname changer
- message sender
- message manager
- file attacher
- member manager
- invitation manager

As invitation manager I want to:

- invite a user to a server in order to let him join the server.

- create an invitation in order to let users join the server.
- choose the expiration date of an invitation in order to manage the invitations.

As a member manager I want to:

- add a role to a member so they can perform specific actions.
- remove a role from a member to prevent them from performing certain actions.
- list the members of a server to manage them effectively.
- temporarily mute members to restrict them from sending messages.
- ban members to prevent them from joining the server.
- kick members to remove them from the server.

As a role manager I want to:

- create a role to define user permissions.
- update a role to modify user permissions.
- delete a role to remove it from the system.
- list the roles of a server to manage them.
- assign roles to members to enable them to perform specific actions.
- remove roles from members to restrict their actions.

As a nickname manager I want to:

- update the nickname of a member to change their display name.
- change my own nickname to update my display name.

As a nickname changer I want to:

- change my own nickname to update my display name.

As a channel manager I want to:

- create a channel to enable users to communicate.
- update a channel to modify its settings.
- delete a channel to remove it from the server.
- list the channels of a server to manage them.
- restrict permissions of user or role on a channel to control user actions.

As a channel viewer I want to:

- list the messages of a channel to view the conversation.
- search for messages in a channel to find specific information.
- list channel of a server to find the channel I want to see the conversation of.

As a message sender I want to:

- send a message in a channel to communicate with other users.
- update a message to correct it.

As a message manager I want to:

- delete a message to remove it from the channel.
- pin a message to keep it visible in the channel.
- perform same action as the message sender.

As a file attacher I want to:

- attach a file to a message to share it with other users.

As a server manager I want to:

- update the server settings to modify its configuration.
- delete the server to remove it from the system.
- perform the same action as the channel manager.

As an administrator I want to:

- perform all actions on the server to manage it effectively.

NOTE: USE CASE DIAGRAM

2. Services

2.1. Users

The user service is responsible for managing user accounts, including registration, authentication, and profile management. It handles user-related operations such as creating, updating, and deleting user accounts. It also manages user preferences, settings, and security features like password resets and two-factor authentication. It will expose a REST API for user management and a gRPC API for inter-service communication.

The user service will rely on Keycloak. In fact all the data will be stored in the Keycloak database. And we will use the Keycloak API to manage the users.

Therefore keycloak will be hold the datas for the all the users of the application. And all the services will enforce the authentication of users through the Keycloak API. It will need to be callable by all the other services in the application in order to allow connection. We will use one keycloak realm for the whole application and create seperate clients if needed for the services. If a service needs to enforce the authentication of a user, it will need to call the Keycloak API to get the public certficate to verify token.

Keycloak will rely on a Postgres database to store the data.

NOTE: SCHEMA OF THE KEYCLOAK DATABASE

```
type User struct {
    ID                string
    Username          string
    ProfilePicture    string
    Email             string
    FirstName         string
    LastName          string
    Password          string
}
```

The user service will also be responsible for managing friendships between users. It will handle friend-related operations such as sending, accepting, and rejecting friend requests. When a friendship is created, a channel will be created between the two users. This channel will be a direct message channel, allowing the two users to communicate with each other. In order to be friend with a user, the user will need to send a friend request to the other user. The other user will be able to accept or reject the friend request. If the friend request is accepted, a channel will be created between the two users. In order to make the friend request, the user will need to know the username of the other user. The username will be unique for each user. The user will be able to search for users by their username. All user should also be able to ask for another user to be friend by cliking on the user profile, for example in a server.

NOTE: SCHEMA OF THE FRIENDSHIP SYSTEM

NOTE: SEQUENCE DIAGRAM FOR FRIEND REQUEST WORKFLOW

NOTE: SCHEMA OF THE DIRECT MESSAGE CHANNEL CREATION

2.2. File storage

The file storage service is responsible for managing file uploads and downloads. It handles file-related operations such as uploading, downloading, and deleting files. It will expose a REST API for file management and a gRPC API for inter-service communication. The file storage service will rely on a Minio server to store the files. Minio is an open-source object storage server that is compatible with Amazon S3. It provides a simple and efficient way to store and retrieve files. The file storage service will use the Minio API to manage the files. It will store the files in a Minio bucket and provide a URL for each file that can be used to access it. NOTE: SCHEMA OF THE MINIO DATABASE

Each service will have its own path to store the files. The file storage service will use a unique prefix for each service to avoid conflicts. For example, the user service will store files in the `user` prefix, while the message service will store files in the `message` prefix.

2.3. Server service

The server service will be responsible for managing servers, members, roles, channels, and webhooks. It will use Postgres to store the data and expose a REST API for server management and a gRPC API for inter-service communication.

This service will also handle writing data to the Permify database. It will be invoked for the following operations involving server members: - Joining or leaving a server - Assigning or removing Server-wide & Application-wide roles - Muting or unmuting members - Banning or kicking members - Inviting or removing members from a server - Assigning roles within a server

Any user can become a member of a server as soon as he is invited to it. An invitation can be sent by any member of the server. The validity of the invitation can be set to multiple ranges:

- 1 hour
- 1 day
- 1 week
- Custom

The service will frequently interact with the Permify service to verify if a member is authorized to perform specific actions.

Direct message channels are independent of servers and allow users to communicate without being part of a server. To interact with a direct message channel, users must be members of the channel. All members of a direct message channel have equal permissions, which cannot be modified.

Channels within a server can have the following types: - **Text Channel**: A basic channel for sending messages. - **Conference Channel**: A voice channel where authorized members can communicate. - **Thread**: A sub-channel within a text channel, created to discuss specific topics without cluttering the main channel. Threads are tied to a message in a text channel. - **Category**: A grouping mechanism for organizing channels within a server.

Authorized members in a server can create roles with various permissions, such as: - Managing the server - Managing roles and assigning them to members - Managing channels - Sending messages - Managing messages (e.g., deleting or pinning) - Viewing channels - Managing webhooks - Managing nicknames - Full server access (inherits all permissions)

Permissions can be assigned to roles, and roles can be assigned to server members. Additionally, channel-specific permissions can override server-level permissions for roles or individual members. These channel-specific permissions include: - Viewing the channel - Managing the channel - Managing webhooks - Managing permissions - Sending messages - Managing messages

To manage these permissions atomically, permission overrides will be stored in the database. A permission override structure might look like this:

```
type PermissionOverwrite struct {
    ID        int      `json:"id"`
    ChannelID string   `json:"channel_id"`
    RoleID    *string  `json:"role_id,omitempty"`
    UserID    *string  `json:"member_id,omitempty"`
    Allow     []string `json:"allow"`
    Deny      []string `json:"deny"`
}
```

The `RoleID` and `UserID` fields can be null, but not both simultaneously.

To enforce rules and list objects for a member, data will be duplicated in both the Postgres database of the server service and the Permify database of the authorization service. This duplication ensures that rules are enforced when members perform actions (e.g., sending a message in a channel) and allows listing objects for members (e.g., listing channels in a server).

Instead of using message queues, direct gRPC calls will be made to the authorization service to replicate data in the Permify database.

The server service will also manage webhooks for servers. A webhook allows third-party clients to send messages to a channel within a server. Webhooks are linked to specific channels and require authentication. Authentication will be handled using a JWT token generated during webhook creation. The token will include the webhook ID and channel ID, signed with a service-wide secret key.

Below is the channel mapping in Go:

```
type Channel struct {
    ID          string `json:"id"`
    ServerID    *string `json:"server_id,omitempty"` // Null for direct message
channels
    Name        string `json:"name"`
    Type        string `json:"type"` // e.g., "text", "conference", "thread",
"category"
    ParentID    *string `json:"parent_id,omitempty"` // Null unless it's a thread or
part of a category
    CreatedAt   time.Time `json:"created_at"`
    Permissions []PermissionOverwrite `json:"permissions"`
}
```

Role can also be scope to the application. This means that the role will be applied to all the servers of the application. This will allow to have a global role that can be applied to all the servers of the application. This will be useful for example for the application administrator that will need to manage all the servers of the application.

2.3.1. Throttling Mechanism

To prevent abuse on the system and ensure the stability of the server service, a throttling mechanism will be implemented. We need to limit users to be part to a maximum of 50 servers. This means a user cannot join more than 50 servers and if he tries to join or create a server, the request will be rejected.

2.4. Messages & Search

The message service is responsible for managing messages in channels. It handles message-related operations such as sending, receiving, and deleting messages. It also manages message history, search functionality, and webhooks for real-time notifications. It will expose a REST API for message management and a gRPC API for inter-service communication. The message service will do not need all lot of relation constraint. It will be able to store the messages in a NoSQL database.

The message service relies on a MongoDB database to store the messages. MongoDB is a NoSQL database that provides a flexible and scalable way to store and retrieve data. It is well-suited for storing messages and allows for efficient querying and indexing.

MongoDB provides rich features for indexing and performing full-text search. The indexation will be done on the file name if the message contains a file and on the content of the message.

```
type File struct {
    ID         primitive.ObjectID `bson:"_id,omitempty"`
    Filename   string             `bson:"filename"`
    Mimetype   string             `bson:"mimetype"`
    Size       int64              `bson:"size"`
    StorageKey string             `bson:"storageKey"`
    UploaderID primitive.ObjectID `bson:"uploaderId"`
    UploadDate time.Time          `bson:"uploadDate"`
}

type Message struct {
    ID          primitive.ObjectID   `bson:"_id,omitempty"`
    SenderID    primitive.ObjectID   `bson:"senderId"`
    ChannelID   primitive.ObjectID   `bson:"channelId"`
    Content     string               `bson:"content"`
    CreatedAt   time.Time            `bson:"createdAt"`
    Attachments []primitive.ObjectID `bson:"attachments"`
    Pinned      bool                 `bson:"pinned"`
    Type        int64                `bson:"type"`
}
```

In the case a user wants to perform a search in one channel, which could be a direct message channel or a server channel, we will only need to filter the messages by the channel id.

In the case a user wants to perform a search in all the channels of a server, we will need to filter the messages by asking all the channels the user has access to on the server. Therefore, with all the searchable channels, we will be able to only filter the messages by the channel ids. In order to limit the number of calls and queries to the database and to other services, we can cache the list of channels the user has access to on the server inside Redis. There are some concerns to have when caching this data inside Redis: - The data can quickly become stale. - The data can be too big to store in Redis.

To address the first concern, we can set a TTL (Time To Live) on the cache. This will ensure that the data is refreshed after a certain period of time. Keeping the data for only 1 minute should be sufficient to keep the data up to date. In fact, the first search request will be slower but should not exceed 1 second.

To address the second concern, we can limit the number of channels that are stored in Redis. We can store only the channels that are used frequently. This will ensure that the data is not too big to store in Redis.

NOTE: Detail the link to the s3 files

Also, messages will be able to hold a link to a file to manage the attachments. This URL will only be a link to the file in the subdirectory dedicated to the message service.

NOTE: Detail the link to the s3 files

Messages can be also sent by the system to notify the users that something happened in a server or in the channel. Can be sent by any service that is authorized to do so. The message will have as sender the user that perform the action. The types of messages that can be sent are:

- message: a simple message
- pinning: a message that is pinned
- thread creation: a thread has been created in the channel
- conference creation: a conference has been created in the channel
- user joined: a user has joined the channel

This will allow to inform the users of the actions that are performed in the channel.

2.4.1. Throttling Mechanism

To prevent abuse and ensure the stability of the message service, a throttling mechanism will be implemented. Without such a mechanism, the service could be easily attacked by sending a large number of requests in a short period, potentially overwhelming the system and degrading its performance.

The throttling mechanism will enforce a limit of 10 messages per user every 10 seconds. This ensures that users cannot flood the service with excessive requests while still allowing legitimate usage.

To implement this, Redis will be used to store the rate-limiting information. Redis is well-suited for this task due to its low latency and support for atomic operations. The following approach will be used:

1. When a user sends a message, the service will check Redis for the user’s message count within the current 10-second window.
2. If the user has already sent 10 messages in the current window, the service will reject the request with a "Too Many Requests" response.
3. If the user has not reached the limit, the service will increment the message count in Redis and allow the request to proceed.
4. Redis keys for rate-limiting will have a TTL of 10 seconds, ensuring that the count resets automatically after the window expires.

This mechanism will ensure fair usage of the service while protecting it from abuse.

```
function isRateLimited(userID, redisClient):
    key = "rate_limit:" + userID
    count = getValueFromRedis(redisClient, key)

    if errorOccurred(count) and errorIsNotKeyNotFound():
        logError("Error checking rate limit")
        return false

    if count >= 10:
        return true

    beginTransaction(redisClient):
        incrementValueInRedis(key)
        setExpirationForKey(key, 10 seconds)
    endTransaction()

    if errorOccurredDuringTransaction():
        logError("Error updating rate limit")

    return false
```

2.5. Authorization

The authorization service will hold the logic that can be used to manage the data in the Permify database. This service will be callable with a GRPC api. Permify leverages a Postgres database to store the data. It will be used to store the roles and the permissions of the users.

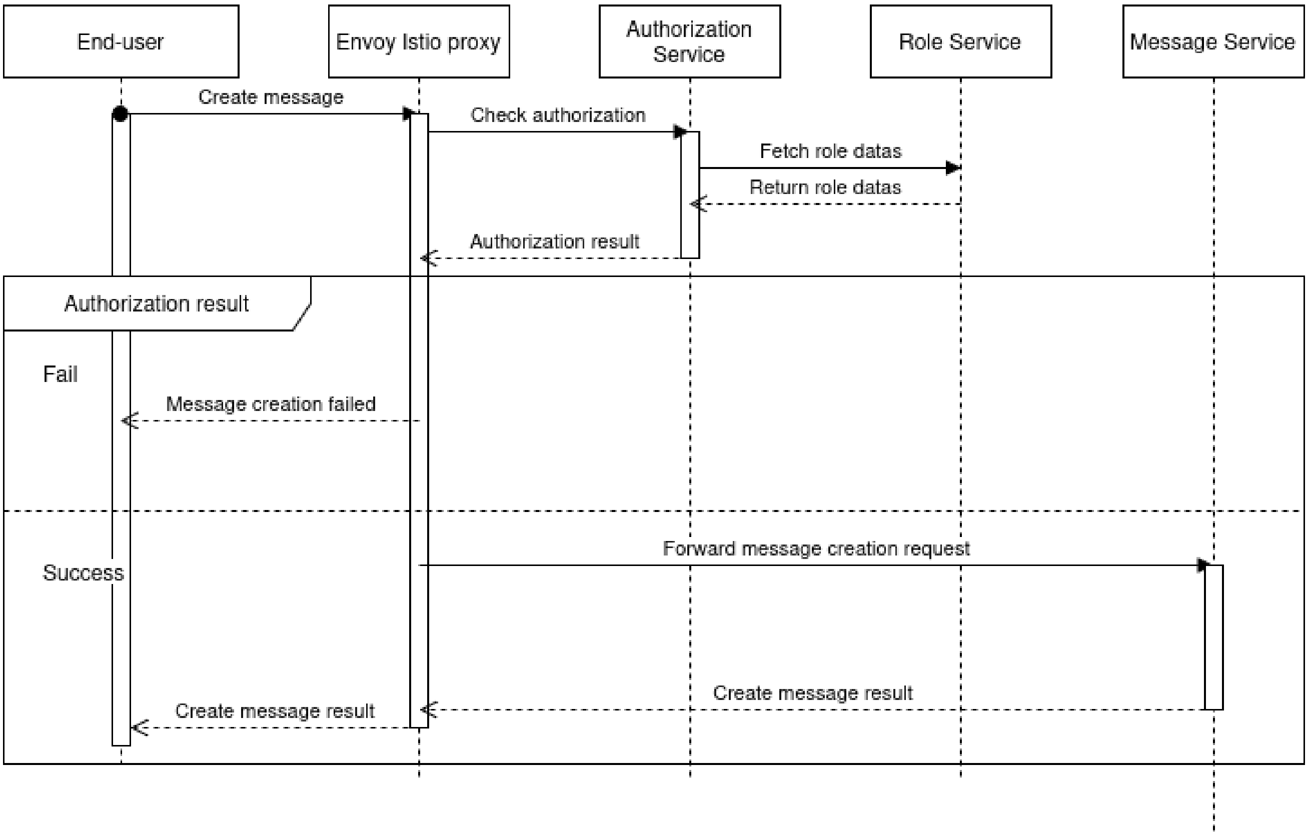


Figure 1. Create a message in a channel of a server and check the authorization

Other services will need to store the data in the Permify database. This service will be called when the following operation are performed. Therefore permify will be able to enforce the rules on the data. Only then all the services in the application will be able to verify requests against the data in the Permify database.

2.6. Voice channel communication

2.7. Schema

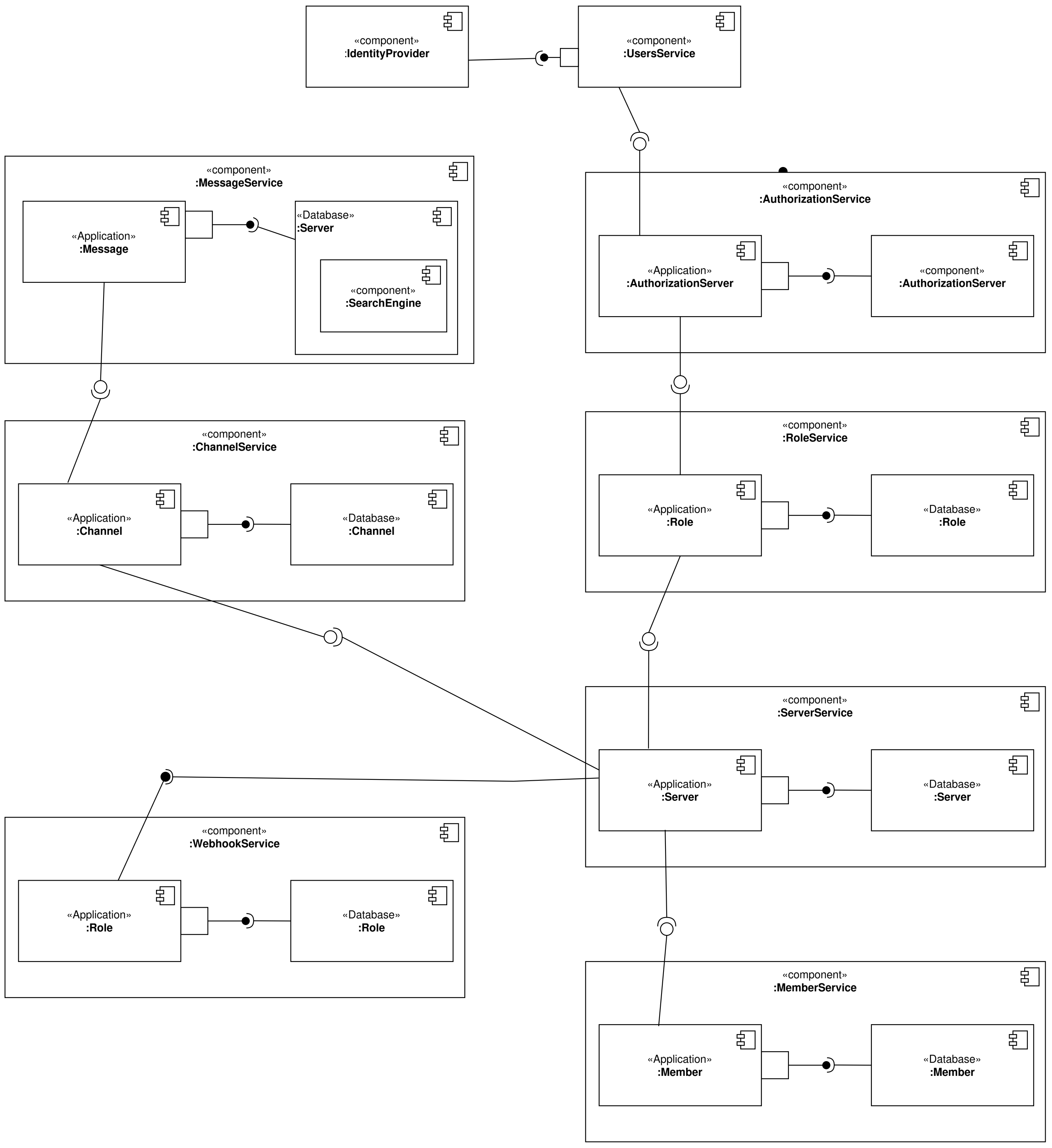


Figure 2. Overview of the application

2.8. Security

2.8.1. Authentication

Authentication is a critical aspect of any microservices architecture. In this document, we will discuss how to integrate Keycloak for authentication to enforce authentication policies at the gateway level.

2.8.2. Keycloak Overview

Keycloak is an open-source identity and access management solution. It provides features such as single sign-on (SSO), user federation, and social login. Keycloak is a suitable choice for our application due to its robust authentication capabilities and ease of integration with microservices.

As the user should be able to authenticate with their email and password, with their google account and their Polytech account from an LDAP Keycloak is suited for this task.

The service allow the user to authenticate natively from frontend implementation by exposing the login page of Keycloak. The user will be able to authenticate with their email and password, with their google account and their Polytech account from an LDAP.

NOTE: SCREENSHOT OF THE KEYCLOAK GOOGLE NOTE: AUTHENTICATION WORKFLOW SEQUENCE DIAGRAM

OAuth2 Overview

OAuth2 is an authorization framework that allows applications to securely obtain limited access to user accounts on an HTTP service by delegating authentication to a centralized identity provider, such as Keycloak.

2.8.3. Get and verify identity

Authorization Code Flow: This flow is suitable for applications that can securely store client secrets. It involves exchanging an authorization code for an access token.

If a user wants to access the frontend of the application without being authenticated, he will be redirected to the Keycloak login page. Once the user is authenticated, the server will issue an access token to the user, which can be used to access the resource. This token is short-lived and can be revoked at any time, providing an additional layer of security.

From the access token the user will be able to access the service. To enforce the check of the access token the service will use the introspection endpoint of the authorization server. Each request will be checked against the introspection endpoint to verify the validity of the token by the istio Gateway.

NOTE: SEQUENCE WORKFLOW FOR THE GATEWAY

We have to note that all service will have an upstream gateway that will check the access token of the user before forwarding the request to the service. This will ensure that only authenticated users can access the services.

Although the user will maybe need to be known by the service, in order to perform some actions. For example, getting the the list of its friends or direct messages. In that case the service will access directly the authorization server to get the user information.

The user will be able to authenticate with their email and password, with their google account and their Polytech account from an LDAP.

2.8.4. Accounts

Via the keycloak frontend the users will be able to manage their accounts. Such as:

- Change their password
- Change their email
- Sign up
- Link their google account
- Link their Polytech account

3. Implementation

3.1. Networking

Microservices implies some networking constraints such as securing a flow of data between services, managing the load of the services, and ensuring the availability of the services.

For that task we will use Istio as a service mesh. It will allow us to manage the networking of the services in a more efficient way.

Poc grpc with rust

3.1.1. Overview of Protocols

Protocols are a fundamental component of microservices architecture, dictating the mechanisms by which services interact and exchange data. This section delves into the technical intricacies of various protocols, including REST, gRPC, and GraphQL, and elucidates the rationale behind selecting gRPC for our application.

HTTP/1.1, commonly used for RESTful APIs, is advantageous due to its simplicity, widespread adoption, and ease of implementation. It supports complex REST APIs and is inherently compatible with web browsers. However, it suffers from several limitations: the lack of type safety, verbosity of JSON payloads, and suboptimal performance due to the overhead of HTTP headers and the text-based JSON format. Despite these drawbacks, REST APIs can be secured using HTTPS with TLS (Transport Layer Security), ensuring encrypted communication.

REST APIs benefit from self-discoverability through OpenAPI specifications, which facilitate seamless integration and collaboration among microservices developed by disparate teams. This discoverability is crucial in a microservices ecosystem where services must interoperate efficiently.

gRPC's strong typing and contract-first approach, enforced through .proto files, ensure consistency and reliability in inter-service communication. This is particularly beneficial in large-scale microservices architectures where maintaining compatibility and preventing breaking changes are paramount.

Given the technical requirements of our application, including the need for efficient, low-latency communication and strong typing, we have chosen gRPC as the primary protocol for inter-service communication. gRPC's performance advantages, coupled with its robust type safety and support for bi-directional streaming, make it an ideal choice for our microservices architecture.

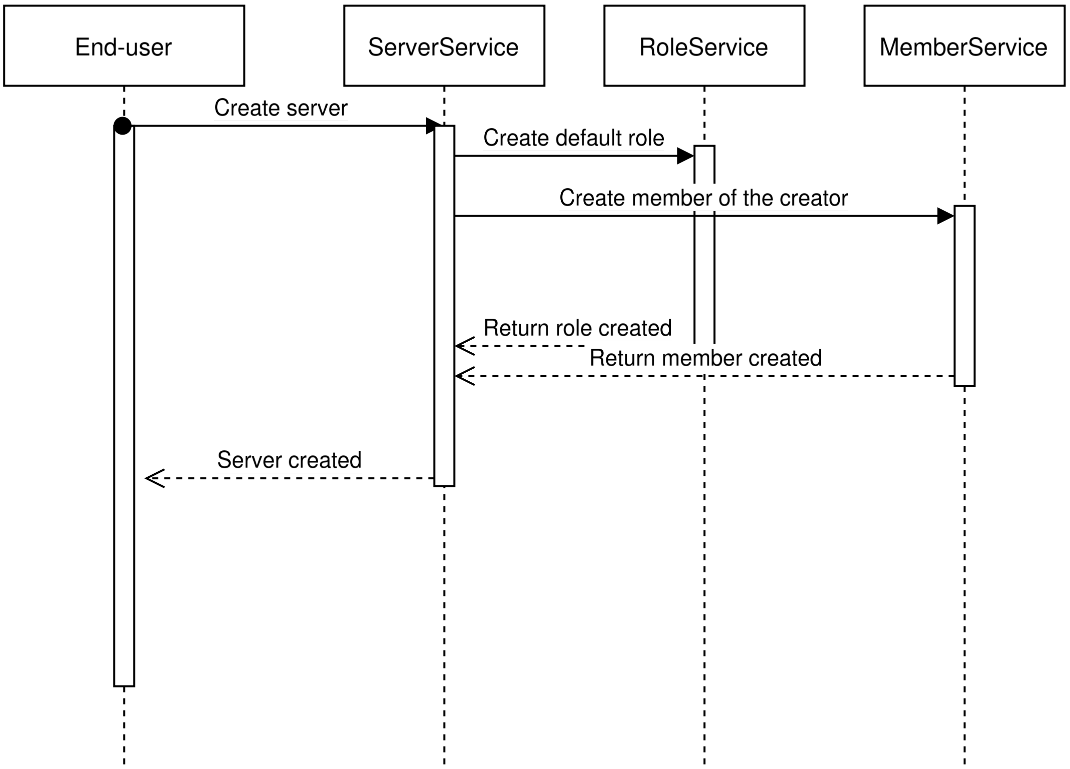
In summary, while REST have its merits, gRPC's technical superiority in terms of performance, efficiency, and type safety aligns with the demands of our application, ensuring reliable and scalable inter-service communication.

3.1.2. gRPC

Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located on another computer in a network. It allows a program to execute a procedure (subroutine) in another address space (commonly on another physical machine). The calling program is suspended until the remote procedure returns, and the remote procedure executes in a different address space. RPC abstracts the communication between the client and server, making it appear as if the procedure call is local.

gRPC is a high-performance, open-source RPC framework developed by Google. It uses Protocol Buffers (protobuf) as the interface definition language (IDL) and leverages HTTP/2 for transport. gRPC offers several advantages over traditional RESTful APIs, including:

- Speed: Faster than REST due to HTTP/2, which allows multiple requests at once, compresses headers, and supports server push.
- Strong typing: Uses protobuf for data, ensuring messages are consistent and efficient.
- Real-time: Supports two-way streaming, letting clients and servers send multiple messages in real-time.
- Multi-language: Works with many programming languages, making it easy to build services in different languages.



3.1.3. mTLS (Mutual TLS)

Mutual TLS (mTLS) is a security protocol that encrypts and authenticates communication between services, ensuring only trusted services can interact. Istio simplifies enabling mTLS across all services in the mesh, enhancing security and preventing unauthorized access.

3.2. Deployment

All services will be deployed across multiple Kubernetes clusters to improve fault tolerance and ensure high availability. Each service will reside in its own namespace, allowing for independent resource management. The clusters will be deployed on Virtual Machines (VMs) to provide flexibility and scalability. We will use Proxmox VE as the hypervisor to manage the VMs, allowing us to create and manage multiple virtual machines on a single physical server.

3.2.1. Kubernetes

The entire infrastructure will run on Kubernetes, divided into two clusters: one for services and another for databases. Services will be isolated within their own namespaces to ensure independent resource management and high availability. Helm charts will facilitate deployment by simplifying configuration, packaging, and dependency handling.

The clusters will be deployed on Virtual Machines with the Talos linux distribution. Talos is a modern, immutable Linux distribution designed specifically for Kubernetes. It provides a lightweight and secure environment for running Kubernetes clusters, making it an ideal choice for our application.

The cluster dedicated for services will expose listen for traffic on port 443 for HTTPS and port 80 for HTTP. It will allow the user to access the web application and the API.

In order to provide a high availability of the application, we will deploy 3 control nodes per cluster which is the recommended configuration by Talos maintainers. The control plane nodes will be responsible for managing the Kubernetes cluster and ensuring that the services are running smoothly.

The worker nodes will be responsible for running the services and the databases. For the databases, we want to ensure a high availability and fault tolerance. We will deploy 3 worker nodes per cluster. This will allow us to have a high availability of the services and the databases. We will see later how to leverage the Kubernetes operators to manage the databases and high availability.

In order to increase the fault tolerance we will use longhorn to manage the storage of the services. Longhorn is a cloud-native distributed block storage solution for Kubernetes. It allows to the replication of the data across multiple nodes. Also it provides tools to backup the kubernetes volumes and to restore them in case of failure.

The cluster holding the databases will will only be reachable from the cluster holding the services. This will ensure that the databases are not exposed to the internet and are only accessible from the services. It means that we have to setup a virtual lan between the two clusters. This will allow the services to communicate with the databases without exposing them to the internet.

In comparaison the cluster holding the services will be reachable from the internet. This will allow the user to access the web application and the API.

All the configuration above will be deployed using a GitOps approach. This means that all the configuration will be stored in a git repository and then deployed using a CI/CD pipeline. This pipeline will be execute a Terraform script that will deploy the clusters. It will allow to have a reproducible and versioned configuration of the clusters. This will ensure that the clusters are always in a consistent state and that we can easily rollback to a previous version if needed.

3.2.2. Networking and Zones

3.2.3. Network Segregation and Security Zones

The infrastructure leverages Proxmox's Software-Defined Networking (SDN) capabilities to create a secure, segmented network environment. This approach allows us to isolate different components of our architecture and control traffic flow between them using virtual networks and firewall rules.

Zone Architecture

Our Proxmox infrastructure is divided into two primary security zones:

1. **Service Zone (DMZ)**: Contains the Kubernetes cluster hosting application services that are accessible from the internet
2. **Database Zone (Secure Zone)**: Contains the Kubernetes cluster dedicated to database services, with no direct internet exposure

This separation follows the defense-in-depth principle, ensuring that database systems are never directly exposed to external networks. Each zone is implemented as a separate VNET in Proxmox SDN, with dedicated subnets and routing configurations.

Table 1. Zone Network Architecture

Zone	Purpose	Network	Internet Access
Service Zone (DMZ)	Hosting user-facing services and APIs	10.0.1.0/24	Yes (controlled)
Database Zone	Hosting database services and persistence layer	10.0.2.0/24	No (isolated)

NOTE: Add a network topology diagram showing the two zones with their respective VNETs, subnets, and connections through firewalls.

Inter-Zone Traffic Control

Traffic between zones is strictly controlled using Proxmox's firewall capabilities. The following matrix defines the allowed traffic patterns between zones:

Table 2. Inter-Zone Traffic Matrix

Source	Destination	Allowed Traffic
Service Zone	Database Zone	- PostgreSQL connections (port 5432) - MongoDB connections (port 27017) - Redis connections (port 6379) - Minio connections (port 9000) - DNS resolution (port 53)
Database Zone	Service Zone	- Monitoring data (port 9090) - Logging data (ports 8125, 24224) - DNS resolution (port 53)
Service Zone	Internet	- HTTPS (port 443) - HTTP (port 80) - NTP (port 123) - DNS (port 53)
Database Zone	Internet	- None (fully isolated)

All other traffic patterns are explicitly denied, creating a least-privilege network environment where only required communications are permitted.

Firewall Implementation

The firewall rules are implemented at multiple levels:

1. **Proxmox SDN Firewall**: Zone-level restrictions controlling traffic between VNETs
2. **Kubernetes Network Policies**: Pod-level restrictions within each cluster
3. **Istio Service Mesh**: Service-to-service communication control

NOTE: Add a diagram showing the firewall architecture with the multiple layers of network security.

Network Path Redundancy

To ensure high availability of network connections between zones, the SDN configuration includes:

1. **Multiple Virtual Bridges**: Each zone has redundant virtual network bridges
2. **Link Aggregation**: Physical network interfaces are bonded for increased throughput and redundancy
3. **Automatic Failover**: In case of link failure, traffic is automatically rerouted

Cross-Zone Service Discovery

For Kubernetes services to communicate across zones, we implement:

1. **Internal DNS Resolution**: CoreDNS provides service discovery between clusters
2. **Istio Multi-Cluster Gateway**: Enables secure service mesh across both clusters
3. **Cross-Cluster Secrets**: Shared TLS certificates for secure authentication

This architecture ensures that while the zones are segregated for security, services can still discover and communicate with each other when explicitly permitted.

Benefits of Zone Segregation

The network zone segregation provides several security and operational benefits:

1. **Defense in Depth**: Multiple layers of security controls protect critical database systems
2. **Reduced Attack Surface**: Database systems have no direct internet exposure
3. **Blast Radius Containment**: Security incidents in the DMZ cannot easily spread to the Database Zone
4. **Independent Scaling**: Network resources can be allocated differently based on the needs of each zone
5. **Simplified Compliance**: Easier to demonstrate security controls for regulated data

By implementing this strict network segregation with granular traffic control, we create a secure foundation for our microservices architecture while maintaining the necessary flexibility for inter-service communication.

3.2.4. Service Mesh Overview

A service mesh is a dedicated infrastructure layer that provides service-to-service communication, observability, and security for microservices applications. It abstracts the network and provides a set of features that simplify the development and operation of microservices.

Service mesh provides the following benefits:

- Traffic management: control the flow of traffic between services, implement routing rules, and perform load balancing.
- Security: provides encryption, authentication, and authorization to secure communication between services.
- Observability: provides metrics, logging, and tracing to monitor the performance and health of services.

3.2.5. Istio Overview

Istio is an open-source implementation of a service mesh that provides advanced networking features for microservices applications. It integrates with Kubernetes and provides a set of tools to manage service-to-service communication, security, and observability.

One of the key advantages of using Istio is that it is actively developed and maintained by a well-known and reputable community. This ensures that the project remains up-to-date with the latest features, security patches, and best practices. As a result, Istio is a reliable and robust choice for a microservices project that is intended to last over time.

One of the main component of Istio is the data plane that will be used to manage the traffic between the services. It will be composed of Envoy proxies that will be deployed alongside the services. All the traffic coming and leaving a pod is redirected to the Envoy proxy that will manage the traffic. This will allow to implement a lot of traffic related features such as load balancing, retries, timeouts, and circuit breaking.

Via proxies Istio is capable to log, trace and monitor natively and seamlessly the traffic between the services. This will allow to have a better observability of the application.

As describe, Istio will allow us to manage a lot of constraints outside of the services and let the services focus on their core functionalities. It is really suitable to delegate the networking constraints to a dedicated service that will manage them in a more efficient way.

3.2.6. Deployment kind

As we will have 2 clusters, one for the services and one for the databases, we will need to deploy Istio in both clusters. Istio manages this case by deploying a control plane in each cluster. It will allow to have the same configuration in both clusters and to manage the traffic between the clusters. This deployment style is called multi-primary and therefore all Istio features will be available between the clusters.

3.2.7. Networking and Security

Istio Gateway is a component that manages inbound and outbound traffic for services in the mesh. It acts as an entry point for external traffic and provides features such as load balancing, routing, and security.

In our application, we will use Istio Gateway to manage external traffic and secure communication with clients. To achieve this, we will integrate Cert-Manager with an external Certificate Authority (CA) such as Let's Encrypt to automate the issuance and renewal of TLS certificates.

Cert-Manager will handle the certificate lifecycle, including requesting, renewing, and injecting certificates into Istio's ingress gateway. By configuring Istio Gateway to use these certificates, we can ensure encrypted communication between clients and services, protecting sensitive data from eavesdropping and tampering.

This approach simplifies certificate management while leveraging a trusted CA like Let's Encrypt to provide secure and reliable TLS for our application.

NOTE: SCHEME OF THE GATEWAY

The gateway will also have the responsibility to verify the access token of the user. This ensures that only authenticated users can access the services behind the gateway. The verification process will involve the following steps:

1. **Token Extraction**: The gateway will extract the access token from the `Authorization` header of the incoming request.
2. **Token Validation**: The gateway will validate the token by calling the introspection endpoint of the Keycloak authorization server. This step ensures that the token is valid, not expired, and issued by a trusted source.
3. **User Information Retrieval**: If the token is valid, the gateway will retrieve user information from the token payload, such as user roles and permissions.
4. **Request Forwarding**: The gateway will forward the request to the appropriate service, including the user information in the request headers for further processing.

This approach centralizes authentication at the gateway level, simplifying the security model for downstream services.

NOTE: SEQUENCE DIAGRAM FOR TOKEN VERIFICATION WORKFLOW

3.2.8. Circuit Breaking

Circuit breaking is a design pattern that prevents cascading failures in distributed systems. It works by monitoring the health of services and breaking the circuit if a service becomes unresponsive or slow.

In our application, we will use circuit breaking to ensure service reliability and prevent service degradation. By implementing circuit breaking in Istio, we can detect and isolate failing services, preventing them from affecting other services in the mesh.

3.2.9. Deploying services

In order to set up the packaging of services, we should follow the principles of 12 factors applications. It will drive us to have a clear separation of concerns between the services and the deployment process.

Each service will be packaged as a Docker image, allowing for easy deployment and scaling. The images will be stored in a private Docker registry to ensure security and control over the deployment process. Then the images will be deployed to the Kubernetes cluster using Helm charts. Helm charts will simplify the deployment process by providing a standardized way to package and deploy applications on Kubernetes. All the charts of the services will be stored inside a registry dedicated to the charts.

We will use a GitOps approach to manage the deployment of services. This means that all the configuration files and Helm charts will be stored in a Git repository. That is why it is important to have convention for the naming of the docker images. We will use the SemVer convention for the naming of the docker images such as `beep-api:1.0.0`.

SemVer Overview

Semantic Versioning (SemVer) is a versioning scheme for software that conveys meaning about the underlying changes. A version number is structured as `MAJOR.MINOR.PATCH`:

- **MAJOR**: Incremented when incompatible API changes are introduced.
- **MINOR**: Incremented when functionality is added in a backward-compatible manner.
- **PATCH**: Incremented when backward-compatible bug fixes are made.

For example: - `1.0.0`: Initial stable release. - `1.1.0`: Adds new features in a backward-compatible way. - `1.1.1`: Fixes bugs without breaking existing functionality.

By adhering to SemVer, we ensure clear communication of changes and compatibility between versions, which is critical for managing microservices in a distributed architecture.

In order to easily integrate services between them the container and therefore helm chart should allow to pass configuration values to the service. The configuration values will be passed to the service using environment variables. This will allow to easily configure the service without having to modify the code. For example, the service will be able to connect to the database using the following environment variables:

```
env:
  - name: DATABASE_HOST
    value: "mongodb://mongo:27017"
  - name: DATABASE_NAME
    value: "beep"
```

3.2.10. ArgoCD

ArgoCD is a declarative, GitOps continuous delivery tool for Kubernetes. It allows us to manage the deployment of applications and services in a Kubernetes cluster using Git as the source of truth. ArgoCD will be used to manage the deployment of services and databases in the Kubernetes cluster. It will monitor the Git repository for changes and automatically deploy the updated configuration to the cluster. This ensures that the deployment process is consistent and repeatable, reducing the risk of errors and improving reliability.

The instance will be deployed in the Kubernetes cluster that will host the databases.

We will store all the configuration values.yaml files of the services in a Git repository. Each services will have its own folder in the repository. The values file will contain the version of the docker image to deploy and the configuration values of the service:

```
image:
  repository: beep-api
  tag: 1.0.0
```

When a service is updated, a pipeline will be triggered to build the Docker image and push it to the Docker registry. The pipeline will also update the values.yaml file in the Git repository with the new version of the Docker image. As ArgoCD is monitoring the Git repository, it will automatically deploy the updated configuration to the Kubernetes cluster.

3.2.11. Deploying databases

We will deploy databases in a separate Kubernetes cluster to ensure isolation and security. Even though it is often advised to deploy databases on bare metal instances, Kubernetes provides a flexible and scalable environment for managing databases. Futhermore, a lot of tools are available to manage databases in Kubernetes. Each database will be deployed using a Kubernetes operator, which simplifies the management and scaling of databases in a Kubernetes environment.

A lot of databases will be used in the application. And they all provide a way kubernetes operator to deploy the database:

- MongoDB Community Operator
- Postgres Operator

- Redis operator
- Redis operator

Each instance of databases will be deployed on it’s own kubernetes node.

In order to scale the databases and allow high availability, we will use the following configuration:

- MongoDB: 3 replicas with sharding enabled
- Postgres: 3 replicas with streaming replication enabled
- Minio: 3 replicas with erasure coding enabled
- Redis: 3 replicas with clustering enabled

For easier management of the infrastructure, Postgres instance will hold multiple databases. Each database will be used by a service. Therefore, the Postgres instance will hold the following databases:

- beep_users
- beep_server
- beep_keycloak
- beep_permify

MongoDB will hold the data for the messages services and the instance of Minio will be used only to store the data for the file storage service.

In order to ensure the integrity of the data, we will use a backup solution to backup the databases. Backups need to be stored in a secure location and should be easily accessible in case of failure. The backups will be stored in a separate S3 bucket serparated from the infrastructure. The kind of backup will change depending on the service we want to backup.

4. Database Backup Strategy

4.1. Recovery Point Objective (RPO) and Recovery Time Objective (RTO)

The defined objectives for our microservices architecture are:

- Messages (MongoDB): RPO of 2 hours
- Critical data (PostgreSQL): RPO of 15 minutes
- All services: RTO of 15 minutes

These values are well-aligned with industry standards for similar applications. For context:

- **Chat applications** typically maintain an RPO of 30 minutes to 4 hours for message data, as temporary message loss is usually acceptable compared to user/account data.

- **Critical user and relationship data** in applications similar to Discord or Slack typically have RPOs of 5-15 minutes to minimize data loss during outages.
- **RTO of 15 minutes** is aggressive but achievable with proper automation and is appropriate for a real-time communication platform where extended downtime significantly impacts user experience.

Incremental backups for MongoDB are indeed possible and well-suited for message data:

MongoDB natively supports incremental backups through its oplog (operations log) This approach is ideal for message data that has high write volume but lower criticality

5. Technical Continuity Plan

5.1. Overview and Strategy

The technical continuity plan ensures our microservices architecture can withstand disruptions and recover quickly from technical failures. This approach focuses on infrastructure resilience, automated recovery procedures, and systematic testing.

The plan addresses infrastructure outages, data corruption, and system failures that could affect our application. Through robust recovery mechanisms and defined procedures, we aim to minimize downtime and data loss when technical issues occur.

NOTE: Add a diagram showing recovery time and recovery point objectives for different system components, with time on the x-axis and illustrating the relationship between the last good backup (RPO) and the time to recovery (RTO).

5.2. Potential Threats and Risk Analysis

Our microservices architecture faces several potential threats that could disrupt normal operations:

- **Infrastructure Outages**: Hardware failures, network disruptions, or power outages affecting clusters
- **Data Corruption**: Database issues from software bugs, hardware failures, or human error
- **Security Breaches**: Unauthorized access, data theft, or malicious attacks
- **Resource Exhaustion**: Traffic spikes, DDoS attacks, or resource leaks causing degradation
- **Configuration Errors**: Misconfigurations during deployments leading to service disruptions

Each threat requires specific mitigation strategies and recovery procedures to minimize downtime.

NOTE: Add a heat map diagram showing the likelihood vs. impact of different threat types, using color coding to highlight high-risk areas that require priority attention.

5.3. Recovery Infrastructure

Our recovery approach leverages the dual-cluster architecture to enable targeted recovery actions. The separation of service and database clusters allows us to recover one cluster independently when the other remains operational.

The technical resilience is built on:

- Infrastructure-as-Code through Terraform for consistent deployment
- Kubernetes operators managing database replication and recovery
- Distributed storage with Longhorn providing data replication
- GitOps deployment with ArgoCD pulling configurations from version control
- Automated backup systems storing data securely off-cluster

This technical foundation creates a system that can rapidly recover from failures with minimal manual intervention.

NOTE: Add an architecture diagram showing the recovery infrastructure components, including clusters, backup systems, and recovery paths. Use color coding to distinguish primary and backup components.

5.4. Recovery Procedures

Each failure scenario requires specialized technical responses:

Database Cluster Failure

When experiencing complete database cluster failure:

1. Monitoring alerts trigger the recovery workflow
2. A replacement cluster is deployed via Terraform in the backup region
3. Database restoration proceeds through specialized operators:
   - PostgreSQL data is recovered using pgBackRest
   - MongoDB collections are restored from snapshots
   - Redis instances are rebuilt (being non-persistent by design)
4. After technical verification, service endpoints are updated via DNS changes

For database corruption scenarios:

1. Write operations are suspended to prevent further data corruption
2. Point-in-time recovery restores to a known-good state
3. Automated data integrity checks validate the recovered data
4. Services resume operations after passing technical validation

NOTE: Add a flowchart showing the decision tree for different database failure scenarios and the recovery paths for each type of database (PostgreSQL, MongoDB, Redis).

Service Cluster Failure

For service infrastructure outages:

1. A replacement service cluster is provisioned via Terraform
2. ArgoCD automatically deploys services from Git repositories
3. Load balancer configurations are updated to route to the new cluster
4. Technical health checks confirm system readiness

For partial service degradation:

1. Circuit breaking isolates failed components
2. Stateless services are restarted with appropriate scaling
3. Stateful services undergo data verification before reactivation
4. System monitors confirm service restoration

NOTE: Add a sequence diagram showing the recovery workflow for service cluster failure, including the interactions between monitoring, Terraform, ArgoCD, and DNS services.

5.5. Testing and Evolution

To maintain technical reliability, we regularly test our recovery mechanisms:

- Simulated cluster failures test full recovery procedures
- Database restoration exercises verify backup integrity
- Controlled chaos engineering identifies resilience gaps
- Network partition tests validate cross-cluster communication

Each test and actual incident triggers a technical post-mortem to identify improvements. This systematic approach ensures our continuity plan evolves alongside our infrastructure.

Our technical documentation includes detailed procedure runbooks, enabling operations teams to follow precise steps during recovery operations. These procedures are version-controlled alongside the infrastructure code itself.

NOTE: Add a diagram showing the continuous improvement cycle for the continuity plan, with phases for planning, testing, evaluation, and improvement.

By treating continuity as a core technical concern rather than an afterthought, we ensure the Beep platform maintains reliability even when facing the inevitable challenges of distributed systems.

6. Monitoring and Observability

Comprehensive monitoring and observability are critical components of our microservices architecture. This section outlines our approach to collecting, processing, and visualizing telemetry data across the platform. We implement the three pillars of observability—logging, metrics, and distributed tracing—to provide a complete view of system behavior and performance.

Our observability stack is built on the Grafana ecosystem, providing a unified experience for visualization and alerting. The entire stack is deployed using the same infrastructure-as-code principles applied to the rest of our platform, with Terraform scripts, Kubernetes operators, and Helm charts ensuring consistent, reproducible deployments.

NOTE: Create a high-level architecture diagram showing the complete observability stack and data flows between components.

6.1. Observability Best Practices

Maintaining a robust observability implementation requires adherence to several key practices. All services are instrumented in a standardized way, ensuring consistency across metrics, logs, and traces. Service Level Objectives (SLOs) are clearly defined for all user-facing services, providing measurable targets for reliability and performance. To prevent metric overload, we carefully manage label cardinality and enforce guidelines for dashboard creation, ensuring that visualizations remain actionable and efficient. Trace context, request IDs, and user context are propagated consistently throughout the system, enabling seamless correlation across observability signals. Regular testing of alerting pathways and data quality ensures that our observability stack remains reliable and effective.

For developers, we provide a streamlined experience with a simplified local observability stack, standardized instrumentation libraries, and comprehensive documentation. This is complemented by custom debugging tools that help correlate metrics, logs, and traces, as well as runbooks that outline procedures for responding to common alerts and issues.

Looking ahead, we plan to enhance our observability stack with automated anomaly detection using machine learning, continuous profiling for performance optimization, and synthetic monitoring to simulate user journeys. We also aim to deepen the integration of business metrics with technical observability and introduce cost attribution features to break down infrastructure expenses by service and team.

6.2. Structured Logging

Our logging strategy uses structured JSON logs for efficient parsing and analysis. All application components emit logs in a consistent JSON format with standardized fields:

```
{
    "timestamp": "2023-10-20T14:30:45.123Z",
    "level": "INFO",
    "service": "message-service",
    "trace_id": "ab23cd45ef67gh89",
    "span_id": "1234567890abcdef",
    "user_id": "user-123456",
    "message": "Message successfully processed",
    "request_id": "req-abcdef123456",
    "additional_context": {
        "channel_id": "chan-123456",
        "message_type": "text"
    }
}
```

6.2.1. Log Collection and Processing

We use the native Kubernetes logging pipeline, where container logs are written to standard output and collected by the Kubernetes node agent (kubelet). These logs are then aggregated and sent directly to Grafana Loki using the Promtail agent, which is lightweight and easy to configure.

A minimal Promtail configuration for Kubernetes:

```
server:
    http_listen_port: 9080
    grpc_listen_port: 0

positions:
    filename: /tmp/positions.yaml

clients:
    - url: http://loki.observability.svc.cluster.local:3100/loki/api/v1/push

scrape_configs:
    - job_name: kubernetes-pods
        kubernetes_sd_configs:
            - role: pod
        relabel_configs:
            - source_labels: [__meta_kubernetes_pod_label_app]
                target_label: app
```

6.2.2. Log Levels

We standardize log levels across all services:

- **DEBUG**: Detailed diagnostic information for development and troubleshooting
- **INFO**: General operational events, successful actions, and state changes
- **WARN**: Unexpected situations that are not errors but may require attention
- **ERROR**: Failures or issues that require investigation and may impact functionality
- **FATAL**: Critical errors causing service termination

All services must use these levels consistently to ensure clarity and actionable logs.

6.2.3. Authorization and Security Logging

For authorization events, we apply special logging rules:

- **INFO**: Successful authorizations, including user, resource, and decision context
- **WARN**: Failed authorizations, with relevant context for security review
- **ERROR**: Repeated or suspicious authorization failures

The authorization service logs: - All access decisions with user, resource, and permission context - Authentication failures (with limited context to avoid information leakage) - Permission and role changes

6.2.4. Log Visualization and Analysis

Logs are stored in Grafana Loki, enabling:

- Efficient, indexed storage for fast retrieval
- LogQL queries for filtering and analysis
- Correlation with metrics and traces in Grafana dashboards

Dashboards include: - Service-level log overviews - Authorization/security event monitoring - Error rate analysis by service and endpoint

6.2.5. Log Retention and Archiving

Log retention policy:

- **Hot Storage**: 7 days for immediate querying
- **Warm Storage**: 30 days compressed for investigation
- **Cold Storage**: 180 days archived in S3-compatible storage for compliance and long-term analysis

6.3. Metrics Collection

Our metrics implementation provides comprehensive visibility into system performance, resource usage, and business operations. We use Prometheus as our metrics collection backend, with Grafana Mimir providing horizontally scalable long-term storage.

6.3.1. System Metrics

We collect detailed metrics from all infrastructure components:

1. **Node-level Metrics**:
    - CPU usage (total, user, system)

    - Memory usage (total, available, cached)
    - Disk usage (capacity, IOPS, latency)
    - Network throughput and error rates
    - System load averages
2. **Kubernetes Metrics**:
    - Pod resource metrics (CPU, memory requests and limits)
    - Pod status and health
    - Deployment/StatefulSet status
    - Node pool capacity and allocation
    - Scheduler and API server latency
3. **Database Metrics**:
    - Connection pool utilization
    - Query performance and error rates
    - PostgreSQL: WAL generation rate, replication lag
    - MongoDB: Oplog size and replication status
    - Redis: Memory fragmentation, keyspace statistics
    - Storage utilization percentage with predictions

6.3.2. Application Metrics

Our services emit detailed metrics about their internal operations and business performance.

1. **General Service Metrics**:
    - Track request rates, error rates, and response latencies for each endpoint.
    - Monitor connection pool and thread pool usage, as well as garbage collection statistics.
2. **Business Metrics**:
    - Measure total active users, server counts, and message throughput.
    - Analyze API usage patterns and error rates by service and type.
3. **Custom SLI Metrics**:
    - Calculate API latency percentiles (such as p50, p95, p99) and end-to-end transaction times.
    - Monitor overall service availability.

All metrics are exposed in a standardized format compatible with Prometheus, using consistent naming conventions across services. This enables unified collection, querying, and visualization of metrics for both operational and business insights.

6.3.3. Service Level Objectives (SLOs)

We define and monitor SLOs for all critical services. Each SLO includes:

1. **Service Level Indicators (SLIs)**: Specific metrics that measure service performance
2. **Targets**: Defined thresholds for acceptable performance
3. **Error Budgets**: Allowed deviation from perfect service
4. **Alerting Rules**: Notifications when approaching or exceeding budget burn rates

Example SLO definition for the message service:

- **SLI**: 95th percentile latency for message delivery
- **Target**: < 500ms for 99.9% of requests in a 30-day window
- **Error Budget**: 0.1% (43.2 minutes of allowed degraded performance per month)
- **Alert**: When 2% of error budget is consumed in 1 hour

6.3.4. Metrics Storage and Visualization

Metrics are stored in Grafana Mimir, which provides:

1. **Horizontal Scalability**: Distributed storage for metrics data
2. **Long-term Retention**: Different retention policies based on metric importance
3. **Multi-tenancy**: Isolation between different components and teams
4. **High Availability**: Redundant storage to prevent data loss

Grafana dashboards provide visualization of these metrics, with specialized views for:

1. **Executive Overview**: High-level system health and business metrics
2. **Operator Dashboards**: Detailed infrastructure and service metrics
3. **Developer Dashboards**: Service-specific performance and debugging views
4. **SLO Dashboards**: Error budget tracking and historical compliance

NOTE: Create a mockup dashboard showing key metrics including active users, message throughput, and service health indicators.

6.4. Distributed Tracing

Distributed tracing provides end-to-end visibility into request flows through our microservices architecture. We leverage Istio’s built-in tracing capabilities, OpenTelemetry for instrumentation, and Grafana Tempo for trace storage and analysis.

6.4.1. Tracing Implementation

Our tracing stack consists of:

1. **Automatic Instrumentation**: Istio injects trace context and generates traces for all service-to-service communication within the mesh.
2. **Manual Instrumentation**: For critical application paths, we use OpenTelemetry SDKs to add

custom spans and attributes.

3. **Trace Context Propagation**: Trace and span IDs are consistently propagated across HTTP/gRPC boundaries, enabling correlation between logs, metrics, and traces.
4. **Sampling Strategy**: Adaptive sampling captures 100% of errors and a representative sample of normal traffic.

Example Istio mesh configuration for tracing:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
    meshConfig:
        enableTracing: true
        defaultConfig:
            tracing:
                zipkin:
                    address: "tempo.observability:9411"
                sampling: 100.0
                custom_tags:
                    environment:
                        literal:
                            value: "production"
```

For enhanced visibility, application code can use OpenTelemetry SDKs (Go, Java, etc.) to manually create spans and propagate trace headers:

```
import (
    "context"
    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "google.golang.org/grpc/metadata"
)

func ForwardTraceHeaders(ctx context.Context) context.Context {
    md, ok := metadata.FromIncomingContext(ctx)
    if !ok {
        md = metadata.New(nil)
    }
    outCtx := metadata.NewOutgoingContext(ctx, md)
    return outCtx
}
```

Example OpenTelemetry Collector configuration to export traces to Tempo:

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: otel-collector-config
    namespace: tracing
data:
    otel-collector.yaml: |
        receivers:
            otlp:
                protocols:
                    grpc:
                    http:
        processors:
            batch:
                timeout: 5s
                send_batch_size: 1024
        exporters:
            tempo:
                endpoint: "tempo:3200"
        service:
            pipelines:
                traces:
                    receivers: [otlp]
                    processors: [batch]
                    exporters: [tempo]
```

6.4.2. Trace Storage and Analysis

Traces are stored in Grafana Tempo, which provides scalable, high-performance storage and integrates seamlessly with Grafana for visualization and analysis. TraceQL enables flexible querying by trace ID, service, operation, or attribute.

Key use cases:

- **Performance Optimization**: Identify bottlenecks in request flows.
- **Error Investigation**: Trace root causes of failures across services.
- **Dependency Analysis**: Understand service interactions and dependencies.
- **User Journey Analysis**: Track end-to-end user interactions.

Example TraceQL queries:

- Retrieve a trace by its trace ID: ` {traceID="abc123"} `
- Find all traces for a specific service: ` {service="my-service"} `

NOTE: Add a sequence diagram showing a sample trace as it flows through multiple services, with annotations for important spans and timing information.

6.5. Alerting

Our alerting strategy ensures actionable alerts, clear ownership, and efficient incident response. Grafana Alerting serves as the central system, integrated with Prometheus (metrics) and Loki (logs).

Alerts are classified by severity (Critical, Warning, Info) and category (Availability, Latency, Error Rate, Saturation, Security, Business). Ownership and escalation paths are clearly defined, including on-call rotations and escalation procedures.

6.5.1. Alert Definition and Classification

Alerts follow a consistent framework:

1. **Severity Levels**:
   - **Critical**: Immediate action required, service impact
   - **Warning**: Needs attention soon, potential future impact
   - **Info**: Notification only, no immediate action required
2. **Categories**:
   - **Availability**: Service/component unreachable
   - **Latency**: Response times exceed thresholds
   - **Error Rate**: Unusual error volume
   - **Saturation**: Resource constraints
   - **Security**: Potential incidents
   - **Business**: Key business metric anomalies
3. **Ownership**:
   - Responsible team and on-call rotation
   - Escalation path for unacknowledged alerts

6.5.2. Notification Channels

Multi-channel notifications ensure timely response:

- **Email**: All alert levels to responsible teams
- **Discord**: Real-time alerts via webhooks to dedicated channels
- **PagerDuty**: Critical alerts trigger on-call notifications
- **Mobile**: Critical and warning alerts to team devices

Grafana's webhook notifier is configured for Discord, with secure storage of webhook URLs and routing based on severity.

6.5.3. Key Alerting Rules

Comprehensive rules include:

- **Infrastructure**: Node CPU/memory/disk >85% for 15m, node not ready for 5m, pod crash loops
- **Database**: Replication lag >30s, disk usage >80% projected in 24h, connection pool >80%
- **Application**: Error rate >0.1% for 5m, 95th percentile latency exceeds SLO for 10m, SLO error

budget >5% in 1h

- **Security**: Multiple failed authentications, unusual API access, bursts of authorization failures
- **Business**: Drop in message throughput, decline in active users, anomaly detection on business metrics

Alerting rules are managed as YAML files and loaded into Prometheus, which continuously evaluates them and sends notifications to configured channels.

6.5.4. Alert Suppression and Correlation

To prevent alert fatigue:

- **Grouping**: Related alerts are grouped
- **Silencing**: Automated during maintenance
- **Correlation**: Identify common root causes
- **Intelligent Routing**: Route to appropriate teams

6.5.5. Alert Management Best Practices

- **Silencing**: Mute alerts during maintenance or known issues
- **Escalation**: Automatically escalate unacknowledged alerts
- **Incident Management**: Integrate with incident tracking tools
- **Post-Incident Reviews**: Identify improvements after major incidents

6.6. Observability Infrastructure

The observability stack is deployed and managed using the same infrastructure-as-code principles as the rest of our platform.

6.6.1. Deployment Architecture

The observability components are deployed across both clusters:

1. **Service Cluster**:
    - Prometheus instances for scraping service metrics
    - Fluent Bit DaemonSet for log collection
    - Istio tracing components
    - Grafana for visualization (primary instance)
2. **Database Cluster**:
    - Prometheus instances for database metrics
    - Fluent Bit for database log collection
    - Central storage components (Loki, Tempo, Mimir)

    - Grafana for visualization (replica)

All components are deployed using Helm charts and managed by Terraform scripts. The observability stack is designed to be resilient and scalable, with redundancy across clusters to ensure high availability.

6.6.2. Resource Requirements

The observability stack is provisioned with appropriate resources to handle the expected load:

1. **Mimir (Metrics)**:
    - Storage: 50GB per day retention, compressed
    - CPU: 8 cores (distributed)
    - Memory: 32GB (distributed)
2. **Loki (Logs)**:
    - Storage: 100GB per day retention, compressed
    - CPU: 8 cores (distributed)
    - Memory: 32GB (distributed)
3. **Tempo (Traces)**:
    - Storage: 20GB per day with 7-day retention
    - CPU: 4 cores (distributed)
    - Memory: 16GB (distributed)
4. **Grafana**:
    - CPU: 2 cores
    - Memory: 4GB
    - Storage: 10GB for dashboard persistence